Ex. 2, Matrices, program flow and sound of music

Stefan Karlsson

January 20, 2014

1 Conditional Stuff

The most basic use of conditional statements is in the `if` construction, which we have used often. A new example:

```
response = input('what is 1+1? (type an answer) : ');
if response == 2
   disp('congratulations, spot on')
end
```

We can get further functionality by using an if-else construction:

```
response = input('what is 1+1? (type an answer) : ');
if response == 2
   disp('congratulations, spot on')
else
   disp('wrong')
end
```

If - elseif - ... elseif - else is a generalization:

```
response = input('what is 1+1? (type an answer) : ');
if response == 2
   disp('congratulations, spot on')
elseif abs(response -2) < 2                  %'abs' is absolute value, so check if answer is close
   disp('wrong, but you are pretty close')
else
   disp('wrong')
end
```

A loops is a piece of code that keep executing, as long as some condition is met. To simulate the behavior of a child on its way to an amusement park:

```
answer = 'No';
while strcmp(answer, 'No')
    pause(1);
    answer = questdlg('Dad... are we there yet?');
end
```

In the above simulation, we can click cancel, or shut down the dialog to exit the conversation. Our loop condition is checking for the answer 'No', thus the answer 'Cancel' will exit the loop. We should instead check for the condition 'NOT yes' (different from 'No'!). We do this by the ~ operator, which we put in the condition of the while loop:

```
answer = 'No';
while ~strcmp(answer, 'Yes')              %notice the   ~   character
    pause(1);
    answer = questdlg('Dad... are we there yet?');
end
msgbox('YAY!!!!');
```

As a proper 8 year old, Matlab will now take nothing but yes for an answer. Canceling the dialog box wont help you.

2 More audio and elementwise operations

Lets continue with our audio example from the last exercise. Recall that you can generate a perfect tone from your speakers using:

```
Fs = 40000;                           %sampling frequency
t  = linspace(0,2,2*Fs);              %2 seconds long vector

y = sin( t*6000);                     %6k frequency
sound(y,Fs);
```

We just produced a tone of 6k frequency, with duration 2 seconds. For making music we use 'notes'. Lets define a note to be a pair of values (a vector of two elements). The first element is frequency, and the second is duration[1].

Provided with this pdf, you have a function file: 'prettyNote1.m', that is designed to synthesize notes. Lets investigate the function `prettyNote1` starting with the syntax:

```
y = prettyNote1(TheNote,bPlay,bDisplay)        %syntax for the function
```

Here is a specification of `prettyNote1`:

- `y`(Audio Vector), synthesized audio.
- `TheNote` (2D vector), information about the note to be synthesized:
  - `TheNote(1)`: The frequency of the tone.
  - `TheNote(2)`: The duration(in seconds) of the sound.
- `bPlay` (boolean[2], default: false), use to playback sound.
- `bDisplay` (boolean, default: false), use to plot audio vectors.

[1] musical notation has more information than this, but these two are the most important
[2] boolean means its a value that is either true or false (0,1)

We will have different versions (`prettyNote1` up to `prettyNote4`).They will share the same syntax, but generate different kinds of sound. The sound of `prettyNote1` is the pure tone, given by:

$$y = \sin(tP)$$

where P is the frequency of the note. The function looks like this:

```
function y = prettyNote1(TheNote,bPlay,bDisplay)
if nargin <2,    bPlay = false;     end     %default values of inputs
if nargin <3,    bDisplay = false;  end
Fs = 40000;                                 %sampling frequency
P = TheNote(1);                             %the frequency of the note
L = TheNote(2);                             %the duration of the note
t  = linspace(0,L,Fs*L);                    %time interval

   %BEGIN COMPUTATION
y  = sin(t*P);                              %pure tone
   %END COMPUTATION

if bPlay
  sound(y,Fs);
  pause(L);
end
if bDisplay
  figure;
  plot(t,y);
  xlim([0   ,0.01]);   %x axis limit to 0.01 seconds
  ylim([-1.5,1.5]);
end
```

To try out our function, we can use it in a few different ways:

```
theNote  =  [4000,1];             %a note of 4k frequency and 1 sec duration
  prettyNote1(theNote,1  );       %play the note
  prettyNote1(theNote,0,1);       %display audio vector of the note
  prettyNote1(theNote,1,1);       %play and display
y=prettyNote1(theNote    );       %gather the audio vector into 'y'
```

The tone doesn't sound very pleasant(pure tones aren't used much in music). To make a nicer sounding tone lets implement a gradual decrease in amplitude over time(sound attenuation). Mathematically:

$$y = \sin(tP) \times \text{lin}(t)$$

where $\text{lin}(t)$ is a function that linearly decreases from 1 to 0

In code, this can be done in two steps: first generate a linearly decreasing vector and then multiply it with the audio. This will be our first example on multiplying vectors together. It is implemented in `prettyNote2`:

```
   %from the function "prettyNote2.m"
   %BEGIN COMPUTATION
tone = sin(t*P);                            %pure tone
lin  = linspace(1,0,Fs*L);                  %linear decrease(attenuation)
y    = tone.*lin;                           %generate the audio
   %END COMPUTATION
```

Try running it with `prettyNote2(theNote,1,1);` There is a little dot in the expression of the last line (`tone.*lin`), and is crucial that we have it.

The annoying elementwise dot

Often one wants to multiply vectors together, take powers of vectors or divide them with each other. There are several ways of doing vector-on-vector operations, so we need to be specific. For now, just keep track of this simple rule: for elementwise multiplication, powers, and division we have to use an additional dot:

```
t1 = [1 2 3];
t2 = [3 2 1];
t1.*t2                  %elementwise multiplication
t1./t2                  %elementwise division
t1.^3                   %elementwise power to '3'
```

You may be curious of what happens if we remove the dot, and write `t1*t2` say, instead of `t1.*t2`. Matlab interprets the `*` as so called matrix multiplication, which is entirely different from elementwise multiplication. *Don't forget the dot when you want to do elementwise operations!* It is the most common and annoying sources of bugs when coding in Matlab.

3 Loops

Recall that we produced some 'music' in exercise 1. Lets do this again here, using our function `prettyNote2`:

```
Fs = 40000;                             %sampling frequency
Bt = 6000;                              %base tone
y1 = prettyNote2([Bt*1,     0.4 ]);
y2 = prettyNote2([Bt*1.115, 0.45]);
y3 = prettyNote2([Bt*0.89,  0.55]);
y4 = prettyNote2([Bt*0.45,  0.4 ]);
y5 = prettyNote2([Bt*0.67,  1   ]);

yTotal = [y1 y2 y3 y4 y5];              %merge(concatenate) the vectors
sound(yTotal,Fs);                       %play the full song
```

If we want the music to never stop, we can put it in a **while** loop. **Warning:** To get Matlab to stop a loop or script you can manually issue a break. Have the command window selected and push **CTRL+C**, or **CTRL+break**.

```
%play some music forever:
while true
  sound(yTotal,Fs);
  pause(2.5);
end
```

We can let the loop run for 3 times instead of forever:

```
p = 1;
while p<4                               %loop as long as p is less than 4
  sound(yTotal,Fs);
  pause(2.5);
  p = p+1;
end
```

An equivalent iteration can be done with the **for** loop:

```
for p = 1:3                              %give the interval of the loop(from 1 to 3)
  sound(yTotal,Fs);
  pause(2.5);
end
```

In both our loop constructs above, `p` is used as a counter. Lets use `p` to incrementally reduce the volume of the audio(the amplitude):

```
for p = 1:3
  disp(['value of p: '  num2str(p) ]);      %display value of p
  sound(yTotal/p,Fs);                       %vary amplitude with p
  pause(2.5);
end
```

Perhaps we would like our playback to reduce volume even more than this. We can change the interval of the loop directly:

```
for p = 1:6:13
  disp(['value of p: '  num2str(p) ]);     %display value of p
  sound(yTotal/p,Fs);                      %vary amplitude with p
  pause(2.5);
end
```

Of course we aren't limited to integer intervals:

```
for p = linspace(1,28,3)
  disp(['value of p: '  num2str(p) ]);     %display value of p
  sound(yTotal/p,Fs);                      %vary amplitude with p
  pause(2.5);
end
```

We can use any reasonable vector to indicate values for `p` to iterate over:

```
for p = [1, 5.5, 89.3]
  disp(['value of p: '  num2str(p) ]);      %display value of p
  sound(yTotal/p,Fs);                       %vary amplitude with p
  pause(2.5);
end
```

Going back to the audio example...

... we have successfully implemented a linear attenuation of the tone (`prettyNote2)`, which has improved its pleasantness somewhat.

To improve it even further, lets add *overtones* to the audio. Here is a script that visualizes the first 2 overtones of a signal:

```
clear;
P = 1;
t = linspace(-pi,pi,5000);
plot(t, sin(t*P  ),   ...          %original signal
     t, sin(t*P*2),   ...          %first overtone...
     t, sin(t*P*3));               %second

ylim([-2,2]);
legend('original','OT 1','OT 2');
```

For our purposes overtones are defined as tones that have frequencies related to the original tone by integer multiplication[3]. The function `prettyNote3` adds one overtone:

```
%from the function "prettyNote3.m"
   %BEGIN COMPUTATION
lin  = linspace(1,0,length(t));       %linear decrease(attenuation)
y  =  0.7*sin(t*P  ).*lin + ...       %tone with linear attenuation
      0.3*sin(t*P*2).*lin.^2;         %1st overtone with attenuation
   %END COMPUTATION
```

appart from just adding the overtone, we also put different attenuation on it (that equals $\text{lin}^2(t)$). To hear this sound, and get both tones(original and overtone) visualized try out:

```
prettyNote3([6000 2],1,1);
```

A tone $\sin(tPk)$, for some positive integer k is an overtone of $\sin(tP)$, and could be attenuated by $\text{lin}^k(t)$ in our synthesizer. Mathematically, we can synthesize as follows:

$$y = \sum_{k=1}^{K} c_k \sin(tPk) \times \text{lin}^k(t)$$

The function `prettyNote3` above, does this for $c_1 = 0.7$, and $c_2 = 0.3$. A sum, of the kind in above equation, is easily implemented in matlab with a `for` loop. In `prettyNote4`, we gather the different weights ('c_k') in the vector `c` as we investigate below:

```
   %code from prettyNote4.m:
   %BEGIN COMPUTATION
c   = [ 0.6    0.1    0.3];
lin = linspace(1,0,length(t));            %linear decrease(attenuation)
y   = c(1)*sin(t*P).*lin;                 %the main tone

for k = 2:length(c)
  y = y + c(k)*sin(t*P*k).*lin.^k;        %summation of overtones
end
   %END COMPUTATION
```

Hopefully, the sound now somewhat resembles that of a plucked string.

4 Matrices

In our examples of playing music so far, we have used single lines for both the definition and the generation of each note, such as:

```
Bt = 6000;
prettyNote4([Bt*0.45,  1 ],1);
```

[3] the overtones we consider are so-called harmonic overtones.

In above example, we generate one note that is 0.45 frequency of the base tone, and is 1 second long. It is convenient to have all the notes and their durations listed in one place. This is exactly what a sheet of musical notes is. One way to implement a sheet of notes is as two vectors: one for the tone, and one for the duration:

```
P  =  [1     1.115   0.89   0.45    0.67]        %the frequency
L  =  [0.4   0.45    0.55   0.4     1   ]        %the duration
```

A nicer way to represent these two vectors is as a single matrix, the sheet:

```
Sheet = [1     1.115   0.89   0.45    0.67; ...
         0.4   0.45    0.55   0.4     1  ]
```

All the information about the n^{th} note is now found in the n^{th} column of matrix `Sheet`. Matrices are indexed in the same way as vectors, but we need one more index for a 2D position in the matrix. For example:

```
Sheet(1,4)                %frequency of the 4th note
Sheet(2,4)                %duration of the 4th note
Sheet(1,1)                %frequency of the 1st note
Sheet(2,1)                %duration of the 1st note
```

To get the width and height of a matrix, we use the `size` function. A sheet always has 2 rows, but the number of columns can vary:

```
[nofRows, nofColumns] = size(Sheet)
nofRows               = size(Sheet,1)       %is always 2 for our Sheet example
nofColumns            = size(Sheet,2)       %is the same as nofNotes
```

As with vectors, we can use indexation to get subsets of the matrix:

```
Sheet(1:end, 4  )         %the 4th note
Sheet(1    ,1:end)        %All the frequencies, of all the notes
Sheet(1:end,1:2:end)      %every second note in the entire sheet
```

The Matlab keyword `end` is used frequently to indicate the end of the index range. It is common to want the full range (`1:end`) so Matlab allows us to shorten this to just (`:`). This makes for a short and easy to read[4] notation:

```
Sheet(:,  4)              %the 4th note
Sheet(1,  :)              %All the frequencies, of all the notes
Sheet(:,  1:2:end)        %every second note in the entire sheet
  %more examples:
Sheet(:,  end:-1:1)       %the entire sheet, in reverse order
Sheet(:,  [1 2 3 2 5 1])  %A new, longer sheet, made from notes in the current one
```

As with vectors, we can use indexation to assign values to subsets of matrices:

```
Sheet(2,  :)       = 1;             %put the duration of all notes to 1
Sheet(1,  1:3:end) = 0;             %frequency of every third note to 0 (silence)
Sheet(2,  :)       = Sheet(2,:)*2;  %double all the durations
```

[4] it also makes the cheerful character combinations "**:)**" and "**(:**" very common in your average Matlab code, whereas "**;)**" is sadly invalid.

Matrices are easily created and manipulated in Matlab, and we will get into that in great detail later. For now, make sure you understand `playSheetOfMusic1` that can read and play musical sheets (available as 'playSheetOfMusic1.m'):

```
function y = playSheetOfMusic1(bPlay,BaseTone,BaseBeat,Sheet)
if nargin < 1,  bPlay = true;    end
if nargin < 2,  BaseTone = 6000; end
if nargin < 3,  BaseBeat = 1;    end
if nargin < 4,
  Sheet = [1    1.115 0.899  0.45 0.675; ...
           0.4  0.45  0.55   0.4  1  ];
end
Fs = 40000;
Sheet(1,:) = Sheet(1,:)*BaseTone;               %scale the frequencies with base tone
Sheet(2,:) = Sheet(2,:)/BaseBeat;               %scale the durations with base beat
nofTones   = size(Sheet,2);                     %nofTones = nofColumns in sheet

   %BEGIN AUDIO GENERATION
y = [];                           %y starts as the empty vector
for k = 1:nofTones
  aNote = Sheet(:,k);
  yNew = prettyNote4(aNote);
  y = [y yNew];                   %concatenate new audio at end of previous audio
end
   %END AUDIO GENERATION
if bPlay
   sound(y,Fs);                      %play the full song
   totDuration = sum(Sheet(2,:));
   pause(totDuration);
end
```

A new command in `playSheetOfMusic1` is found at the end of the function:

```
totDuration = sum(Sheet(2,:));
```

As you may guess, `sum` outputs the sum of all vector elements. In the code, this corresponds to the sum of all the notes duration, giving the total duration of the sheet. Lets try the music at 2 different base tones:

```
playSheetOfMusic1(1,6000);
playSheetOfMusic1(1,4000);
```

Lets change the beat:

```
playSheetOfMusic1(1,6000,2);
playSheetOfMusic1(1,6000,0.5);
```

Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

Task 1

The following defines a shape that is like the character ∞:

```
t = linspace(-pi,+pi,100);
x = cos(t);
y = sin(2*t);
plot(x,y);
xlim([-1.5,1.5]);
ylim([-1.5,1.5]);
```

If we want to animate a small particle moving in this trajectory:

```
for p = 1:length(x)
   plot(x(p),y(p), 'o');
   xlim([-1.5,1.5]);
   ylim([-1.5,1.5]);
   pause(0.05);
end
```

Your task is to implement a function `e2_1.m`:

```
e2_1(resolution,bTrace)
```

When called, `e2_1.m` will produce animation along the ∞ path with one important difference from above: *The particle will be drawn as red instead of blue, when it occupies the lower half of the ∞ shape.* Specification:

- `resolution`(positive integer), number of frames in the animation(sample points on the path).
- `bTrace`(boolean), if True, the particle should leave a trace of itself(its previous positions should not be removed when a new position is plotted).

Hand in only the function file `e2_1.m`.

See next page for hints on this task

Hints for task 1

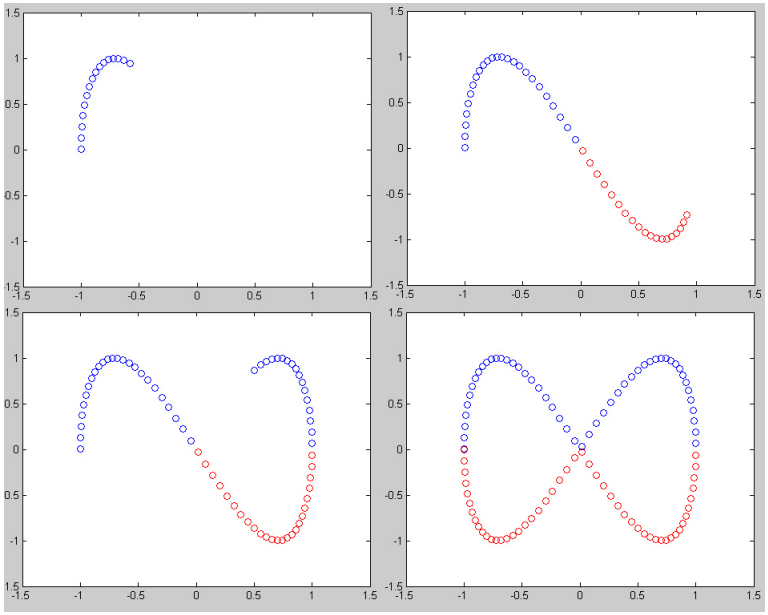

Figure 1: output

- The output from `e2_1(100,100,1)` is visualized in fig. 1, for 4 different times during the same execution.
- What does `hold on` do?

Task 2

Make a new function `e2_2.m` which will work as a new 'prettyNote' function (it will synthesize sound, and share the same syntax as described in section 2). The function `e2_2.m` should add a so-called *tremolo* effect. Mathematically, we want to do the following:

$$y_{new} = y_{old} \times \frac{1 + \sin(t\frac{P}{90})}{2}$$

where y_{old} is the audio as generated in `prettyNote4`, i.e. with attenuation and overtones effects.

Simply put, tremolo is the effect of sound periodically increasing and decreasing in amplitude over time.Tremolo is often mistaken for vibrato, which is a rapid change in frequency, not amplitude. The idea is to make a secondary oscillating function: $\frac{1+\sin(tP/M)}{2}$, of much lower frequency than the pure tone.

Task 3

Make a script `e2_3.m` that plays music using the sound from `e2_2.m`. The music played should be the one defined by the sheet:

```
Sheet = [1    1.115 0.899  0.45 0.675; ...
         0.4  0.45  0.55   0.4  1  ];
```

The sheet should be played twice, first at a base tone of 6000, then at a base tone of 4000.

During playback, there should be the sound of birds chirping playing simultaneously with the music. The sound of birds should be heard twice at the first note each time you play the sheet.

4.0.1 Hints

- the audio file 'e2output.wma' provided, will let you hear what the correct output sounds like. (A script that loads this file, is not considered a correct solution)
- The sound of birds chirping can be loaded into the workspace through:

  ```
  load chirp
  ```

  Be warned that this loads a variable `y`, and a variable `Fs`, that will replace any variables of those names.

- The commands:

  ```
  sound(y1,Fs1);
  sound(y2,Fs2);
  ```

  plays two audio vectors simultaneously, while the commands

  ```
  sound(y1,Fs1);
  pause(2.7);
  sound(y2,Fs2);
  ```

  First plays `y1`, waits 2.7 seconds before starting to play `y2`.